CHAPTER 7

G

7.1 *Introduction to Graphics Programming*

7.2 *Summary*

In this chapter

Since the advent of *Windows*, computer graphics has been assumed as a feature of a computer. Before that it was a relatively rare thing, relegated to some research, to a few expensive Hollywood movies, and to science fiction. Believe it or not, the first use of computer graphics in a commercial motion picture was in the film *The Andromeda Strain* (1971) in which it was used to show a rotating 3D map (they called it an *electronic diagram*) of the underground installation where the action mainly takes place. A few years later the film *Westworld* (1973) used 2½ minutes of digitally processed video to show the visual perspective of an android. It was a very time-consuming and expensive task at that time; it took about 8 hours to process 10 seconds of film, or about 120 hours in all.

Modern computers all possess very fast graphics cards that perform most of the rendering tasks, and added to the built-in software on current operating systems, it allows for a very sophisticated yet simple-to-use graphical/windows interface to desktop computers. The graphics software is hierarchical; the screen itself is merely an array of picture elements (*pixels*) that can be set to any color, and it is difficult to see how that can be made to display complex pictures.
It has reached the point where everything seen on a computer screen is actually drawn—icons, windows, backgrounds, and even text.

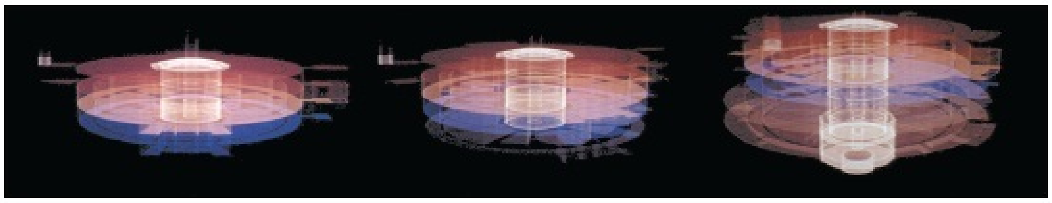


Figure 7.1
Stills from the first computer graphic sequence in a major motion picture: *The Andromeda Strain* (Courtesy of Universal Studios Licensing LLC).

What this means is that interacting with a computer is now done with graphics, not characters and text. Since that is the situation, it makes sense to permit a beginning programmer to experiment with programming graphics applications.

7.1 INTRODUCTION TO GRAPHICS PROGRAMMING

At the most primitive level of graphics software is the ability to set individual pixels. It is, as was mentioned, quite difficult to use this capability to create complex pictures. How is a dog drawn, or a building, or even just a straight line? Those things have been figured out, fortunately.

So at the bottom layer of software are functions that manipulate pixels. At the next level are lines and curves; these are the basic components of drawings and sketches. An artist with a pencil uses lines and curves to represent scenes. At the level above lines are functions that use lines to create other objects, such as rectangles, circles, and ellipses. These can be line drawings or can be filled with colors. The next higher levels can be argued about, but text is probably in the next software layer and then shading and images followed by 3D objects, which includes perspective transformation and textures.

Python does not itself have graphics tools, but various modules that are associated with Python do. The standard graphical user interface library for use with Python is *tkinter*. There are many features of this module, including the creation of windows, drawing, user interface widgets like buttons, and a host of other features. It is free and is normally included in the Python distribution but it can easily be downloaded and used with any Python version. Because there are many ways that Python can be configured on various different systems, the installation process will not be described in detail here. A graphics module will be described and is included on the DVD that accompanies this book, and it requires that *tkinter* be available. This is almost always true (remember that this book uses Python 3).

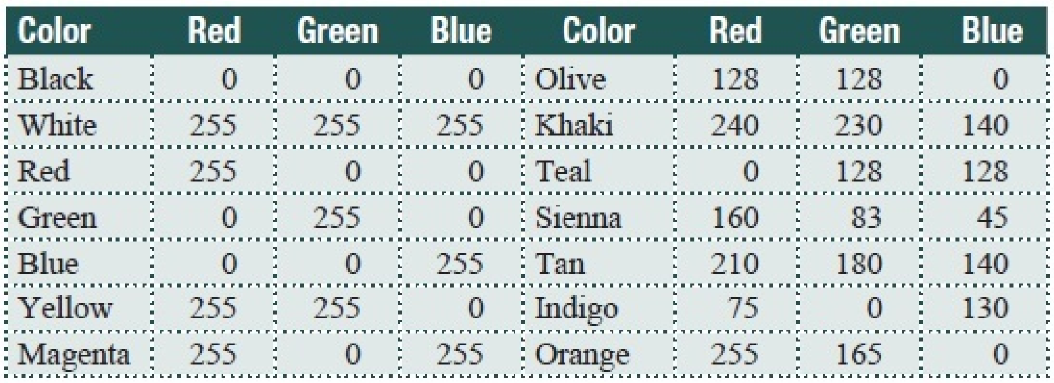
The library that allows graphics programming is called *Glib*. If the *Glib.py* file is in the same directory as the source code for the Python program that uses it, then it should work fine. *Glib* consists of a collection of functions that implement the first few levels of a graphics system and that create a window within which drawing can take place. It does not allow for interaction, animation, sound, or video, all of which are the subject of the next chapter.

7.1.1 Essentials: The Graphics Window and Colors

To start creating computer graphics, it is necessary to understand how colors and images are represented. When using a computer everything must be represented as numbers. A pixel is the color of a picture at a particular location, and so there must be a way to describe a color at that place. In physics frequency is used: each color has a specific frequency of electromagnetic radiation. Unfortunately, this does not map very well onto a computer display, because it is based on television technology. On a TV, there are three colors, red, green, and blue, that are used in various proportions to represent every color. There are red, green, and blue dots on the TV screen that are lit up to various degrees to create the colors that are seen. This is based on the way a human eye sees color; there are red, green, and blue sensors in the eye that in combination create our color perception. Another reason that frequency is not used is that there are colors that are not accurately represented as frequencies; they do not appear in the rainbow. The colors pink and brown are two examples.

So, each color in the graphics system is represented as the degree of red, green, and blue that combine to create that color. In that sense it is a bit like mixing paint. Yellow, on a computer, is a mixture of red and green. Each pixel therefore has three components: a red, green, and blue component. These could be expressed as percentages, but when using a computer it is better to select numbers between 0 and 255 (8 bits, or 1 byte) for each color. Each pixel requires 3 bytes of storage; actually 4 bytes in some cases, as will be seen shortly. If an image contains 100 rows of 100 pixels, then it has 10,000 pixels and is 10000*3=30000 bytes in size.

To humans, colors have names. Here's a list of some named colors and their RGB equivalents:



Color	Red	Green	Blue	Color	Red	Green	Blue
Black	0	0	0	Olive	128	128	0
White	255	255	255	Khaki	240	230	140
Red	255	0	0	Teal	0	128	128
Green	0	255	0	Sienna	160	83	45
Blue	0	0	255	Tan	210	180	140
Yellow	255	255	0	Indigo	75	0	130
Magenta	255	0	255	Orange	255	165	0

There are, of course, a great many more named colors, and even more colors that can be represented with RGB values in this way—16,777,202 of them in fact. Each pixel is a color value. All grey values have the special situation R=G=B, so there are 256 distinct values of grey ranging from black to white.

The graphics library provides functions for creating colors. The functions are **cvtColor()** and **cvtColor3()**:

cvtColor(g) - Return a color value that has R=G=B = g, which is to say it specifies a grey level = g.

cvtColor3(r,g,b) - Return a color value that has the specified R,G,B values.

When using *Glib* the program must initialize the system before drawing anything. All graphics must take place between a call to **startdraw()** and a call to **enddraw()**. If **startdraw()** is not called then important items will not be initialized, most especially a drawing window will not be created and an error will occur at some time during execution. If **enddraw()** is not called then nothing will be drawn. For an abstract example, a hypothetical main program could be:

```
# start of program
    Many calculations
   . . .
   startdraw(width, height)
```

```
# Graphics calls
    . . .
     enddraw()
```

The function **startdraw()** accepts two parameters: the width and the height of the drawing region to be created. These values can be retrieved by a program using calls to the functions **Width()** and **Height()** if they are needed. **Startdraw()** creates the needed window and drawing area, initializes starting colors, fonts, and modes, but *does not open the window*. Nothing that is drawn will be visible until **enddraw()** is called.

7.1.2 Pixel Level Graphics

The only pixel level operation draws a pixel at a specified location; so, for example, the call:

```
point (x, y)
```

will set a pixel at column (horizontal position) **x** and row (vertical position) **y** to a color. What color? *Glib* has two default colors that can be set: the *fill* color, which is the color with which pixels will be drawn, polygons and ellipses will be filled, and characters will be drawn; and the *stroke* color, the color with which lines will be drawn. Setting the fill color is done using a call to the **fill()** function:

```
fill (200) # Set the fill color to (200, 200, 200)
fill (100, 200, 100) # Set the fill color to (100, 200, 100)
```

Using one parameter sets the fill color to a grey value, whereas passing three parameters specifies the red, green, and blue values (respectively) of the fill color. Similarly the function **stroke()** can accept one or three parameters:

```
stroke (200) # Set the stroke color to (200, 200, 200)
stroke (255, 0, 0) # Set the stroke color to
# (255, 0, 0) = red
```

There is one more function that could be thought to be in the pixel level category. The function **background()** is used to set the background color of the graphics window. Again, it can accept either one parameter (grey) or three
(color). This leads to the first example.

Example: Create a Page of Notepaper

Notepaper has blue lines separated by enough space to write or print text between them. It often has a red vertical line indicating an indentation level, a place to begin writing. Drawing this is a matter of drawing a set of connected blue pixels in a set of rows, and then making a vertical column of red pixels. Here is one way to code this:

```
from Glib import *
y = 60 # Height at which to start
startdraw(400, 600) # Begin drawing things
background(255) # Paper should be white
fill (0, 0, 200) # Fill color = pixel
# color = blue
for n in range (0, 27): # Draw 30 horizontal blue
# lines
for x in range (0,Width()): # Draw all pixels in one
# line
point (x, y) # Draw a blue pixel
y = y + 20 # The next line is 20
# pixels down
fill (200, 0, 0) # Pixel color red
for y in range (0, Height()): # Draw connected vertical
# pixels
point (25, y) # to form the margin line
enddraw()
```

The output of this program is shown in Figure 7.2a. When pixels are drawn immediately next to each other they appear to be connected, and so in this case they form horizontal and vertical lines. This is not easy to do for arbitrary lines; it is not obvious exactly which pixels to fill for a line between, say, (10, 20) and (99, 17). That's why the line drawing functions exist.

Example: Creating a Color Gradient

When creating a visual on a computer, the first step is to have a clear picture of what it will look like. For this example, imagine the sky on a clear day. The horizon shows a lighter blue than the sky directly above, and the color changes continuously all of the way from horizon to zenith. If a realistic sky background were needed, then it would be necessary to draw this using the tools available. What would the method be?

First, decide on what the color is at the horizon (y=ymax) and at the highest point in the scene (y=ymin). Now ask: "how many pixels between those points?" The change in pixel color will be the color difference from ymax to ymin divided by the number of pixels. Now simply draw rows of pixels beginning with the
horizon and moving up the image (i.e., decreasing Y value), changing the color by this amount each time.

As an implementation, assume that the color at the horizon will be blue = (40, 40, 255) and the top of the image will be (40, 40, 128), a darker blue. The height of the image will be 400 pixels; the change in blue over that range is 127 units. Thus, the color change over each pixel is going to be 127.0/400, or about 0.32. A color can't change a fractional amount, of course, but what this means is that the blue value will decrease by 1 unit for every 3-pixel increase in height. Do not forget that the horizon is at the bottom of the image, which has the greatest Y coordinate value, so that an increase in Y means a decrease in height and vice versa.

The example program that implements this is:

```
from Glib import *
blue = 128
startdraw(400, 400)
delta = 127.0/Height()
for y in range (0, Height()):
    yy = Height()-y
    fill(40, 40, blue)
    for x in range(0, Width()):
      point (x, y)
    blue = blue + delta
enddraw()
```

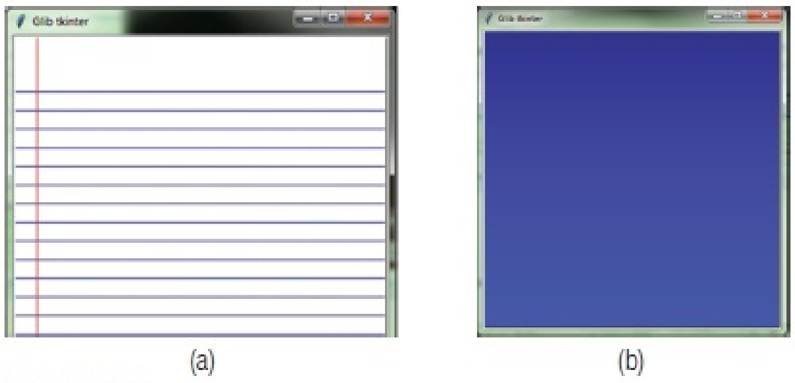


Figure 7.2
(a) A graphic of a sheet of lined paper; (b) a color gradient.

Figure 7.2b shows what the gradient image looks like (a full-color version of this and all images is on the accompanying disc).

7.1.3 Lines and Curves

Straight lines and curves are more complex objects than pixels, consisting of many pixels in an organized arrangement. A line is actually drawn by setting pixels, though. The fact that a

line() function exists means that the programmer does not have to figure out what pixels to draw and can focus on the higher level construct, the line or curve.

A line is drawn by specifying the endpoints of the line. Using *Glib* the call is:

```
line ( x0, y0, x1, y1)
```

where one end of the line is at (x0,y0) and the other is at (x1,y1). The color of the line is specified by the stroke color. If any part of the line extends past the boundary of the window, that's OK; the line will be clipped to fit.

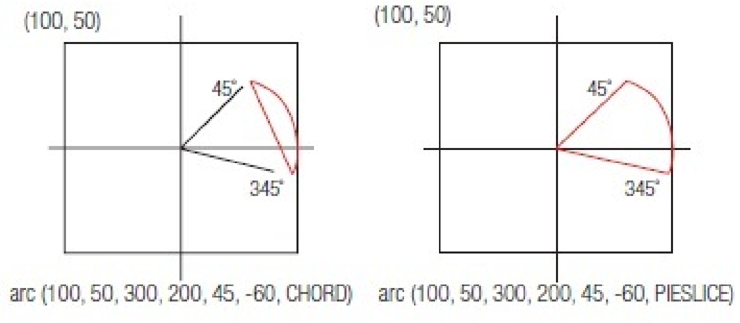
Example: Notepaper Again

The example of drawing a piece of notepaper can be done using lines instead of pixels, and will be a little faster. Set the stroke color to blue and draw a collection of horizontal lines (i.e., that have the same Y coordinate at the endpoints) separated by 20 pixels, as before. Then draw a vertical red line for the margin. The program is a variation on the previous version:

```
from Glib import *
y = 60 # Height at which to start
startdraw(400, 600) # Begin drawing things
background(255) # Paper should be white
stroke (0, 0, 200) # Fill color = pixel
# color = blue
for n in range (0, 27): # Draw 30 horizontal blue lines
line (0, y, Width(), y) # Draw a blue horizontal line
y = y + 20 # The next line is 20 pixels
# down
stroke (200, 0, 0) # Pixel color red
line (25, 0, 25, Height()) # Draw a vertical red line
enddraw()
```

The output from this program is the same as that for the version that drew pixels, which is shown in Figure 7.2a.

A *curve* is trickier than a line, in that it is harder to specify. The method used in Glib is based on that in *tkinter*: a curve (arc) is defined as a portion of an ellipse from a starting angle for a specified number of degrees, as referenced from the center of the ellipse. The angle 0 degrees is horizontal and to the right; 90 degrees is upwards (decreasing Y value). The ellipse is defined by a bounding rectangle, specifying the upper left and lower right coordinates of a box that just holds the ellipse. So, looking at Figure 7.3, the rectangle defined by the upper left corner at (100, 50) and the lower right at (300, 200) has a center at (200, 125) and contains an ellipse slightly longer than it is high (upper left of the figure). The function that draws a curve is named **arc()**, and takes the upper left and lower right coordinates, a starting angle, and the size of the arc also expressed as an angle.



(100, 50)
45°
345°
arc (100, 50, 300, 200, 45, -60, CHORD)
(100, 50)
45°
345°
arc (100, 50, 300, 200, 45, -60, PIESLICE)

Figure 7.3
The result of calls to the arc function with various parameters. This illustrates how the function can be used.

In the upper right of the figure is the arc drawn by the call **arc(100,50, 300, 200, 0, 90)**, which means that the part of the ellipse from the 0 degree point *counterclockwise* for 90 degrees will be drawn. The example at the lower left of the figure draws the curve from the 45-degree point for 90 degrees, resulting in the upper section of the ellipse being drawn. The final arc shown, at the lower right, uses a negative angle. The call **arc(100,50,300,200,45,-60)** starts at the 45-degree point and draws the arc *clockwise*, because the angle specified was negative.

This way of specifying arcs is fine for simple examples and single curves, but makes combining many arcs into a more complex curve rather difficult. Joining the ends together smoothly is the trick.

The arc() function has another parameter that can be specified. The seventh parameter tells the system what kind of arc to draw; the possibilities are ARC, CHORD, and PIESLICE, and the default value ARC is illustrated in the figure. The call:

arc (100, 50, 300, 200, 45, -60, CHORD)

gives the result shown in Figure 7.3a. The result of the call:

arc (100, 50, 300, 200, 45, -60, PIESLICE)

is shown in Figure 7.34b. If filling is turned on, then these shapes would be filled with the default fill color. A major example involving curves/arcs is the pie chart program later in this chapter.

7.1.4 Polygons

For the purposes of discussion, a polygon will include all closed regions, including ellipses and circles. In that context, the **arc()** function can be used to draw circles and ellipses by specifying an angle very near to 360 degrees. The call:

```
arc (100, 100, 300, 300, 0, 359.9, ARC)
```

will draw a nearly complete circle, one that looks complete. However, there is a function that draws real circles and real ellipses: it is named **ellipse()**. In the default drawing mode the **ellipse()** function is passed the coordinates of the center of the ellipse and its width and height in pixels. If the width and height are equal, then the result is a circle. The call:

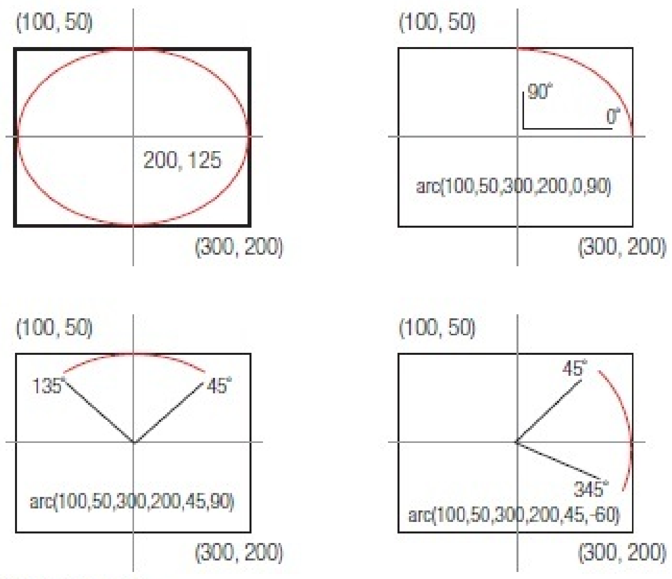


Figure 7.4
The CHORD parameter (left) and the PIESLICE parameter (Right) to the `arc` function.

```
ellipse (100, 100, 30, 40)
```

draws the ellipse shown in Figure 7.4a. The color with which the ellipse is filled is the fill color. If no filling is desired then a call to **nofill()** turns off filling.

The default mode for drawing ellipses is referred to as CENTER mode, where the center of the ellipse is given. There are three others: RADIUS mode, in which the width and height parameters represent semi-major and semi-minor axes; CORNER mode in which the upper left corner is specified instead of the center; and CORNERS mode, in which the upper left and the lower right corner of the bounding box are specified. The mode is changed using a call to the function **ellipsemode**(), passing one of the four constants as the parameter: CENTER, RADIUS, CORNER, or CORNERS.

A rectangle is drawn using the **rect()** function. Again, there are four drawing modes. Consider the call **rect (x, y, w, h)**:

CENTER mode: x, y are the coordinates of the center; w and h are the width and height.

RADIUS mode: x, y are the coordinates of the center; w and r are the horizontal and vertical distances to an edge.

CORNER mode: x, y are the coordinates of the upper left corner; w and h are the width and the height.

CORNERS mode: x, y are the coordinates of the upper left corner;
w and h are the coordinates of the lower right corner.

As with the ellipse, the rectangle drawing mode is changed by call a function, this time **rectmode()**.

The function **triangle()** draws—yes, a triangle. It is passed three points, which is to say six parameters: triangle (10,10, 20, 20, 30, 30) draws a triangle between the three points (10,10), (20,20), and (30,30).

7.1.5 Text

Drawing text is not complicated, but changing fonts is more of a problem.
A font is saved on a file and has to be installed. If a font is specified by a program but does not exist, then either the finished image will look different from what was anticipated or an error will occur. Drawing text is performed by a call to text():

```
text ("Hello there.", x, y)
```

This draws the text string "Hello there!" in the graphics window so that the lower left of the text is at location x, y. The default text size is 12 pixels, but can be changed by a call to **textsize()** passing the size desired.

7.1.6 Example: A Histogram

A histogram is a way to visualize numerical data. It is especially useful for discrete data like colors or political parties or choices of some kind, but can also be used for continuous data. It displays counts of something against some other value, a category; percentage of people voting for specific parties, or the heights of grade six girls. It draws bars of various heights each representing the number of entries in each category. In this example the only problem is the plotting of the histogram, but the more general programming problem would include collecting and organizing the data. In this case the program will read a data file named "histogram.txt" that contains a few key values. The program variable names and the corresponding data file values are:

Variable	Contents
title	Title to be drawn at the top of the graph
ncategories	Number of categories
maxsize	Maximum size of any category
hlabel	Horizontal label
vlabel	Vertical label
val[1]	Value for category 1
val[2]	Value for category 2
	...
lab[1]	Label for category 1
lab[2]	Label for category 2
	...

When creating a graph, it pays to design it carefully. In this case the histogram will have the general appearance shown in Figure 7.5a. This visual layout helps with the details of the code, especially if the design has been drawn on graph paper so that the coordinates can easily be determined.

Assume that the variables needed have been read from the file (See: Exercise 2). Here's what the program must do:

```
Create a window about 600x600pixels in size.
Draw the horizontal and vertical axes (120, 80)
Draw the title and axis labels.
```

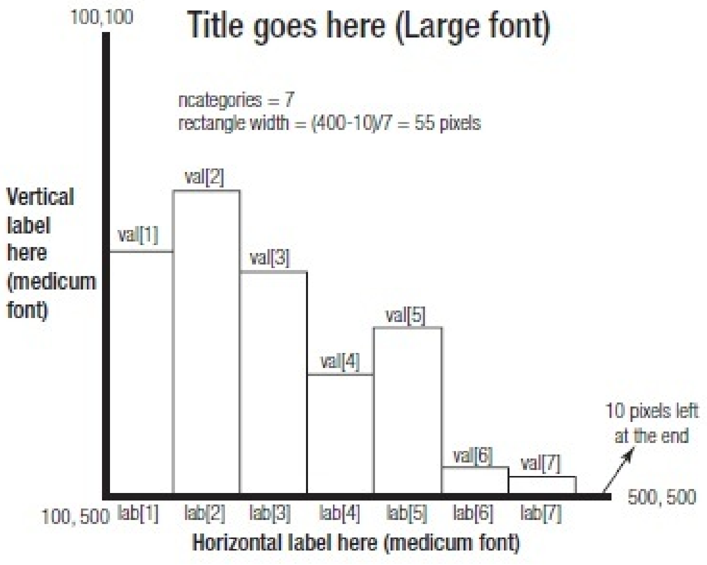

100,100
Title goes here (Large font)
ncategories = 7
rectangle width = (400-10)/7 = 55 pixels
Vertical label here (medicum font)
val[1]
val[2]
val[3]
val[4]
val[5]
val[6]
val[7]
10 pixels left at the end
500, 500
100, 500
lab[1]
lab[2]
lab[3]
lab[4]
lab[5]
lab[6]
lab[7]
Horizontal label here (medicum font)

Figure 7.5
The visual design of a histogram before it is coded.

```
Determine the width and height of each rectangle.
For i in range (0,ncategories)
```

```
        Draw rectangle i
        Draw label i
```

Development can now proceed according to the plan. Create a window, set the background, and change the font to *Helvetica* (a favorite):

```
startdraw (600, 600)
background(180)
setfont("Helvetica")
```

Now draw the axes. Y-axis (vertical) from 100,100 to 100,500; X-axis (horizontal) from 100,500 to 500,500. Use a thicker line by setting the stroke weight to 3 pixels.

```
  strokeweight (3)
  line (100, 100, 100, 500) # Y axis
  line (100, 500, 500, 500) # X axis
```

The title is in a large font (24 pixels) at the top part of the canvas (y=80)

```
textsize (24) # Title uses a big font
text (title, 120, 80) # It's at the top of the drawing
```

The horizontal axis label is in a smaller font (14 pixels) at the bottom of the canvas (y-580). It looks nicer if the text is basically centered. It's hard to do this exactly because there is no easy way to find out how long a string will be when drawn. However, a string that is 14 pixels in size and N characters long will be approximately N*14 pixels long when drawn. The axis is 400 pixels wide, so the amount of leftover space will be 400-N*14. Divide this in half to get the indentation of the left to approximately center the string!

```
textsize (14) # Labels use a medium sized font
cx = (400-len(hlabel)*14)/2 # How many pixels to indent
text (hlabel, 100+cx, 580) # Centered at the bottom
```

Drawing the vertical label is more difficult, and so it will be done later. Make it a function and move on

```
verticalLabel(vlabel)
```

It is time to draw the rectangles. The width of each one will be the same, and is the width of the drawing area divided by the number of categories. The height will be the height of the drawing area divided by the maximum value to be drawn, **maxsize**. Compute those values and set the line thickness to one pixel, then set a fill color. Using the CORNER rectangle drawing mode is easiest for this application, as all that needs to be figured out is the upper left coordinate—the width and height are already known.

```
wid = (400-10)/ncategories # Width of a box in pixels
ht = 390.0/maxsize # Each value is this many pixels high
```

```
strokeweight (1) # Rectangle outline 1 pixel thick
fill (200, 50, 200) # Purple fill
rectmode (CORNER) # Corner mode is easiest
```

Now make a loop that draws each rectangle. The X position of a rectangle is its index times the width of a rectangle—that's easy. The height of the rectangle is the value of that data element multiplied by the variable **ht** that was determined before. It's also a good idea to draw the value being represented at the top of the bar, which is just above and to the right of the rectangle's upper left.

```
for i in range(0,ncategories):
   ulx = 100 + i*wid+2 # Upper left X
   uly = 500 - val[i]*ht # Upper left Y
   rect (ulx, uly, wid, val[i]*ht-2) # Height is val[i]*ht
   text (val[i], ulx+5, uly-2) # Draw the value at the top
```

Finally, draw the labels for each rectangle. These are below the X axis, centered more or less within the horizontal region for each bin. The labels start at the Y axis (X=100 or so) and their location increases by the width of the bin during each iteration of the drawing loop. The Y location is fixed, at 520—the X axis is 500. Finally, an attempt to center these labels is done in the same way that it was done for the horizontal label, but the parameters are different.

```
x = 100+2 # Start at X=100 with extra for the thinck line
textsize(10) # Use a small size font
fill (255, 255, 255) # Text will be white
for i in range (0,ncategories): # for each rectangle
    cx = (wid-len(lab[i]*9)) # Indexnt to center the text
    if cx < 0: cx = 0 # Indent can't be negative
    text (lab[i], x+cx/2, 520) # Draw the label
    x = x + wid Next label is one rectangle right
enddraw()
```

Drawing the vertical label involves pulling out the individual words and drawing each one on its own pixel row. Words are separated by spaces (blanks), so one way of drawing the vertical text is to look for a space in the text, draw that word, then move down a few pixels, extract the next word, draw it, and so on until all words have been drawn. The text will be drawn starting at X=12, and the initial vertical position will be 200, moving down (increasing Y) by 40 pixels for each word. This is done by the function **verticalLabel()**, which is passed the string to be drawn:

```
def verticalLabel(v):
    lasti = 0 # Index of the next word in the string
    x = 12 # Start drawing at X=12
    y = 200 # and Y=200
for i in range(0, len(v)): # Look at all characters in
# the label
if (v[i] == " "): # A black indicate the end of a
# word
text (v[lasti:i], x, y) # Draw the word
y = y + 40 # Move down 40 pxiels
lasti = i # end of this word is the
# startof the next
    text (v[lasti:], x, y) # Draw the last word
# after exiting the loop
```

This program is available on the disk as two examples: “viperHisto.py” and “gradesHisto.py,” which draw histograms of two different sets of data. The output from these two programs is shown in Figure 7.6.

This is a minimal program, and won’t always create a nice image. Labels that are too long and use too many categories can cause badly formatted graphics.

7.1.7 Example: A Pie Chart

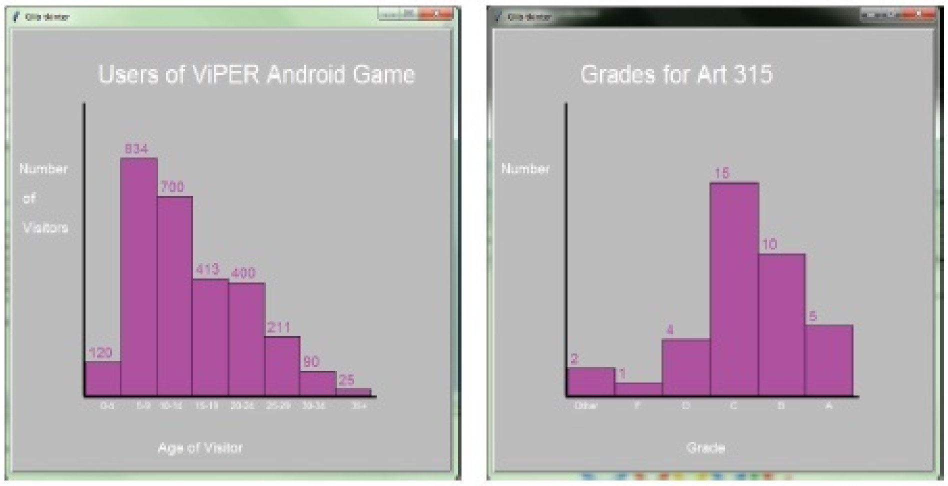
A pie chart is really just a histogram where the relative size of the categories is illustrated by an angle instead of the height of a rectangle. Each class is shown as a pie-shaped slice of a circle whose area is related to its proportion of the whole sample. Pie-shaped regions are easy, because **arc()** will draw them. So, using the same examples as before, look specifically at the grades data: there are 38 students whose grades are being displayed, and there are 360 degrees in a circle. A category of 10 students, for example (such as those receiving a “B” grade), will represent a pie slice that is 10/38 of the whole circle, or about 95 degrees. The process seems to be to determine how many degrees each category represents and draw a pie slice of that size until the whole pie (circle) is used up.

```
Create a window about 600x600 pixels in size
Draw the title label
Establish a fill color
For i in range (0,ncategories)
    Determine the angle A used for this category i
    Draw arc from previous angle for A degrees
    Draw label i for this slice
    Change the fill color
```

The labels may present a problem, as they may not fit inside the pie slice. It is probably best to display the label outside of the slice and draw a line to the slice that represents it.

The program is similar to that for the histogram. So, beginning after the label is drawn:

Find the total number of elements in all categories—in this case, the number of students in the class. This is the sum of all elements in **val.**



(a) (b)

Figure 7.6
(a) Histogram of a set of data on access to a cell phone game;
(b) grades in a university art course.

```
totalSize = 0
r = 255
fill (r, 200, 200)
for i in range (0, ncategories):
    totalSize = totalSize + val[i]
```

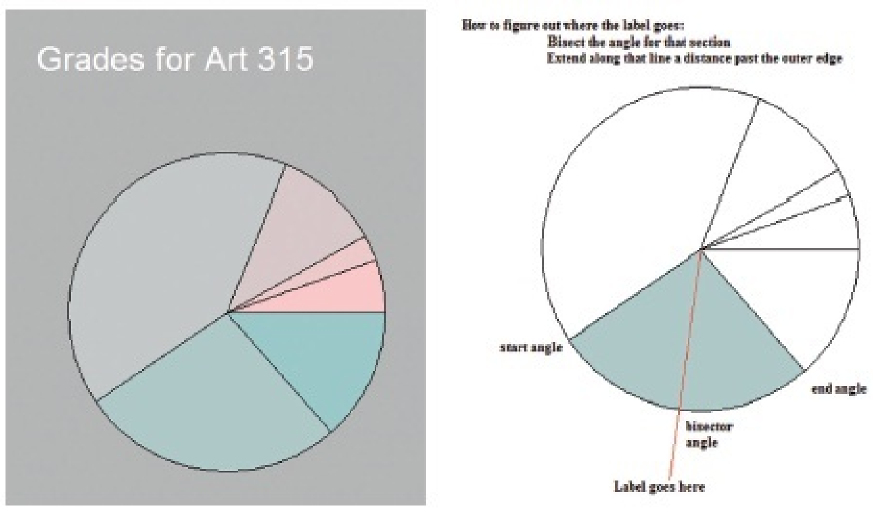
Each count val[i] in a category represents **val[i]/totalSize** of the entire data set, or the angle **360.0*val[i]/totalSize**. The constant **360/totalSize** will be named **anglePerCount**. Now starting at angle 0 degrees, create a pie-shaped arc the size of each category:

```
angle = 0
for i in range(0,ncategories):
    span = val[i]*anglePerCount
    arc (150, 150, 450, 450, angle, span, PIESLICE)
```

Draw the label—this has been left for later, so call a function.

```
label (300, 300, 150, lab[i], angle, span)
```

The angle to start drawing must be increased so that the next arc starts where this one left off:



(a) (b)

Figure 7.7
(a The pie chart drawn using Glib, (b) how to find the placement for labels on the chart.

```
angle = angle + span
```

Change the fill color so that each pie piece is a different color. The code below changes the red component just a little.

```
  r = r - 20
  fill (r, 200, 200)
```

Figure 7.7b shows a way to determine where a label could go; a line from the center of the circle through the outer edge points in the direction of the label. Simply find the x and y coordinates. The y coordinate is the sine of the angle * the distance from the center, and the x coordinate is the cosine of the angle * the same distance. For a distance, use the radius * 1.5. The function **label()** can now be written:

```
def label (xx, yy, r, s, a1, ap):
   angle = a1 + ap/2 # Bisector= start angle + half of
# span
   d = r*1.25 # Distance
   x = cos (angle*3.1415/180.) * d + xx # Angle is radians
   y = -sin (angle*3.1415/180.) * d + yy # Y is inverted
   text (s, x, y)
```

The result is illustrated in Figure 7.8.

There’s one more thing that could be added to the pie chart program. Sometimes, one of the pieces is moved out of the circle to emphasize it. It turns out that this useful feature can be implemented in a manner very similar to the way the labels were drawn. Find the bisector of the angle for that section and before it is drawn identify a new center point for that piece a few pixels down that bisector. This pulls the piece away from the original circle center.

The code is brief, and is included below to be complete:

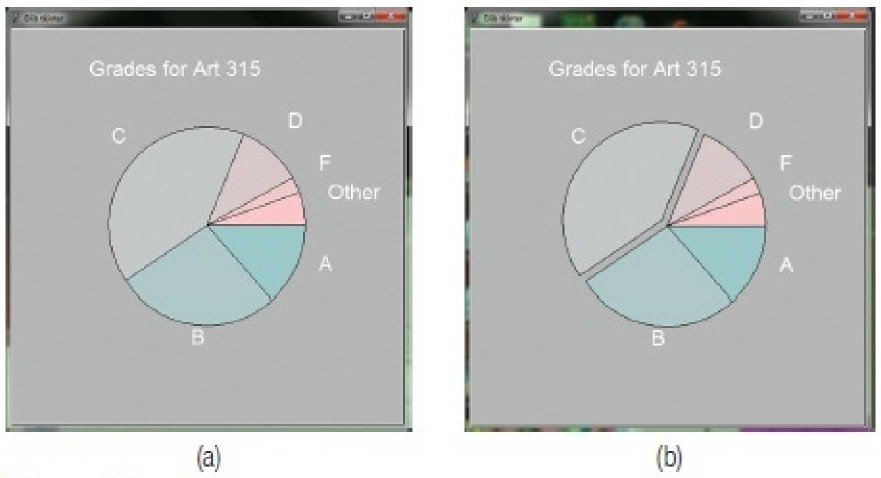


Figure 7.8
(a) The basic pie chart with labels;
(b) the chart with one of the sections separated to emphasize it.

```
def pull (x, y, a1, ap):
   angle = a1 + ap/2 # a1 is the angle, ap is the span
   d = 12 # Pull out only a small amount
# (12 pixels)
   y = -sin (angle*3.1415/180.) * d + y # New center
# coordinates
   x = cos (angle*3.1415/180.) * d + x
   arc (x-150, y-150, x+150, y+150, a1, ap, PIESLICE)
```

This function is called instead of the call to **arc()** for the piece that is to be pulled out. A sample output is shown in Figure 7.8b.

7.1.8 Images

Unlike the graphical components displayed so far, an *image* is fundamentally a collection of pixels. A camera captures an image and stores it digitally as pixels, and so it was never anything else. Displaying an image means drawing each pixel in the appropriate color, as captured.

Glib can load and display images in a limited fashion. Images reside in files of various formats: JPEG, GIF, BMP, PNG. The same image in each format is stored in a quite distinct way, and it can require a lot of code just to get the pixels from the image. Python, through *tkinter*, allows GIF image files to be read directly. Any image file can be converted into a GIF by using one of a hundred conversion tools, including *Photoshop*, *Paint*, and *Gimp* to name a few.

The function **loadImage()** will read a GIF image file and return an image of a sort that can be displayed in the graphics window. The file “charlie.gif ” is a photo of checkpoint Charlie in Berlin, and has been included on the accompanying disc. It could be read into a Python program with the call:

```
im = loadImage ("charlie.gif")
```

The variable **im** now holds the image, and while the details are not completely relevant, it is good to know that **im.width** and **im.height** give the width and height of the image in pixels. Displaying the image is a matter of calling the function image() and passing the image and the coordinates where the upper left corner of the image is to be placed. A call such as:

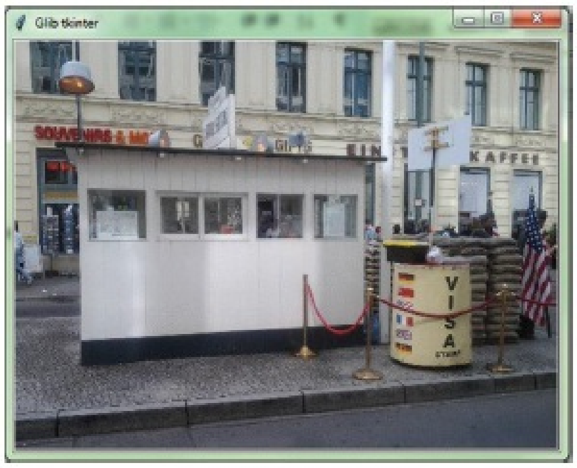


Figure 7.9
Original test image – Checkpoint Charlie in Berlin.

```
image (im, 0, 0)
```

would display the checkpoint Charlie image.

The smallest Python program (using *Glib*) that can load and display an image is thus 5 lines:

```
from Glib import *
startdraw(600, 600)
im = loadImage ("charlie.gif")
image (im, 0, 0)
enddraw()
```

However, this displays the image in a window that is bigger than it is. That may be OK, but often an image is to be used as a background image, and the size of the window should be the same as the image size. If that is what is needed there is a special function to call:

imageSize(). It must be called before **startdraw()**, of course, because it returns the size of the image, and hence the size to be passed to **startdraw()**. It returns a tuple that has the width as the first component and the height as the second.

Opening a window that is the same size as an image is accomplished as follows:

```
from Glib import *
s = imageSize("charlie.gif")
startdraw(s[0], s[1])
im = loadImage("charlie.gif")
image (im, 0, 0)
enddraw()
```

The output of this program is shown in Figure 7.9.

Pixels

An image as returned by **loadImage()** is a built-in type named *PhotoImage*. It is really not designed to be used for much except displaying in the window, but some more advanced operations are possible, if slow. For example, individual pixel values can be extracted and modified. *Glib* offers a handful of low-level functions to make pixel operations easier.

An image consists of rows and columns of pixels, and a pixel is a color. The color of the pixel at horizontal position **x** and vertical position **y** of image **pic** can be accessed using the call:

```
c = getpixel(pic, x, y)
```

The value of **c** is a color, and the components of this can be accessed using the functions:

```
r = red(c)
g = green(c)
b = blue(c)
```

Setting the value of the pixel at location (x,y) is accomplished by calling setpixel(). It requires that the image, the coordinates, and the color be passed as parameters:

```
setpixel (pic, x, y, c)
```

These functions operate on an image, not directly on the screen display. Changes to an image will only be visible if they are done before the image is displayed.

Example: Identifying a Green Car

There is a pattern here that is important to recognize when working with images at the pixel level—the raster scan. All of the pixels in the image are usually examined one at a time using a nested loop. It will look like this:

```
  for i in range(0, Width()):
      for j in range(0, Height()):
  # Do something to pixel (i,j)
```

This example uses color to identify the pixels that belong to a car in an image, as seen in Figure 7.10. The problem requires identifying pixels that are “green” and somehow making them stand out in the image. What is green? All pixels have a green component. When something is green, the green component is the most significant one; it is larger than the red and blue components by some margin. In this case that margin will be arbitrarily set at 20, and if it does not work then it can be modified. If a pixel is green it will be set to black, otherwise it will become white; this will make the pixels that belong to the car stand out.

The program begins by creating a window and reading in the image:

```
startdraw(640, 480)
im = loadImage ("eclipse.gif")
```

Now look at all of the pixels, searching for a green one:

```
for i in range(0, Width()):
    for j in range(0, Height()):
c = getpixel(im, i,j) # Get the color of the pixel
# (i,j)
```

If the pixel is green, then change it to black. Otherwise, change it to white:

```
if green(c)>(red(c)+20) and green(c)>(blue(c)+20): # Green?
setpixel (im, i, j, cvtColor3 (0,0,0)) # Black
else:
    setpixel (im, i, j, cvtColor3(255,255,255)) # White
```

An image can be saved into a file by calling the *Glib* function **save()**:

```
save (im, "out.gif")
```

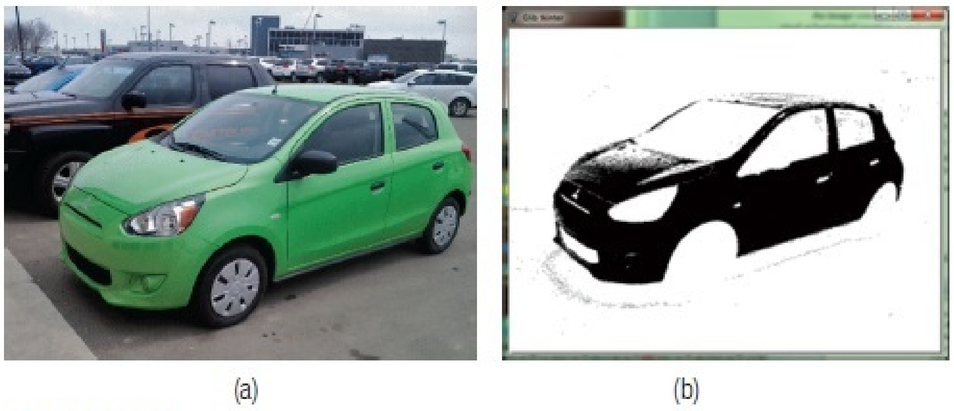


Figure 7.10
(a) A green car; (b) the result of changing green pixels to black and all others to white.

The output of this program is shown in Figure 7.10b. There are some “green” pixels that do not belong to the car, but most of the car pixels have been identified.

Example: Thresholding

Image processing is a large subject all by itself, and this particular library is not the best choice for exploring it in detail. There are some basic things that can be done, and common ones include thresholding, edge enhancement, noise reduction, and count, all of which can be done using *Glib*. Thresholding in particular is an early step in many image-analysis processes. It is the creation of a bi-level image, having just black and white pixels, from a grey or color image. The previous example is different from thresholding in that a particular color was being searched for. In thresholding a simple grey value T, the *threshold*, is used to separate pixels into black and white: all pixels having a value smaller than T will be black, and the others will be white.

Glib has one more function that is useful, especially in this context. The function **grey()** will convert a color into a simple grey level, which is an integer in the range 0 to 255. It finds the mean of the three color components. The thresholding program begins in the same way as did the previous example. Look at the color of all of the pixels in the image, one at a time:

```
startdraw(640, 480)
im = loadImage ("eclipse.gif")
for i in range(0, Width()):
    for j in range(0, Height()):
        c = getpixel(im, i,j)
```

This is the standard scan of all pixels. Now convert the color c to a grey level and compare that against the threshold T=128. Pixels having a grey level below 128 will be set to black, the remainder will be white:

```
    if g < T:
        setpixel (im, i, j, cvtColor3 (0,0,0))
    else:
        setpixel (im, i, j, cvtColor3(255,255,255))
image (im, 0, 0)
save (im, "out.gif")
enddraw()
```

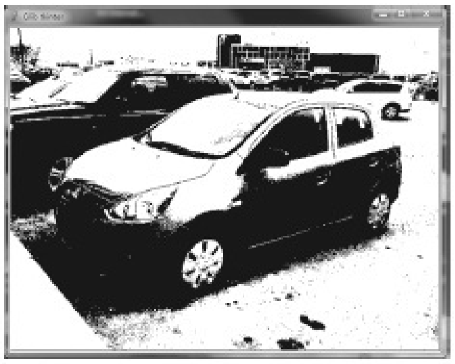


Figure 7.11
Thresholded version of the green car image of Figure 7.10

The result, the image displayed by this program, is shown in Figure 7.11.

Transparency

A GIF image can have one color chosen to be transparent, meaning that it will not show up and any pixel drawn previously at the same location will be visible. This is very handy in games and animations. Images are rectangular, whereas most objects are not. Consider a small image of a doughnut; the pixels surrounding it and in the hole can have the pixels set to be transparent, and then when the image is drawn over another one the background will be seen through the hole.

The transparency value must be set within the image by a program. Photoshop, for example, can do this. Then when Python displays the images, the background image must be displayed first, followed by the images with transparency. As an example, Figure 7.12a shows a photo of the view through the rear and side windows of a Volvo. The window glass area, the places where transparency is desired, have been colored yellow. The color yellow was then selected in Photoshop as transparent, and the image was saved again as a GIF. A short Python program using *Glib* will display a background image and the car image over it, and the background will be seen through the window regions as in Figure 7.12b. The program is:

```
from Glib import *
s = imageSize("car.gif") # Get size of image
startdraw(s[0], s[1]) # Open window of the right
# size
s = loadImage("perseus.gif") # Background image
t = loadImage ("car.gif") # Car image with transparency
image (s, 0, 0) # Display background first
image (t, 0, 0) # then display the car image
enddraw()
```

7.1.9 Generative Art

In *generative art* an artwork is generated by a computer program that uses an algorithm created by the artist. The artist is the creative force, the designer of the visual display, and the computer implements it. There are many generative artists to be found on the Internet: one list can be found here:

http://blog.hvidtfeldts.net/index.php/generative-art-links/

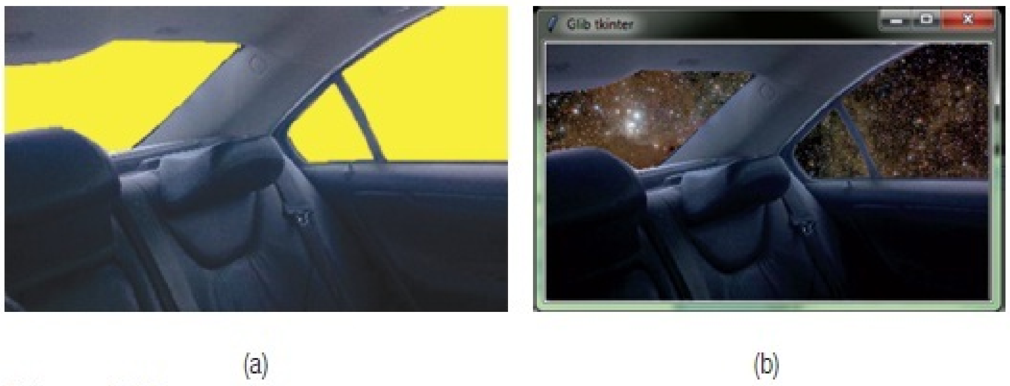


Figure 7.12
(a) An image of a car interior. The window areas have been edited manually to be some color that does not appear in the image otherwise. This color is then set to be transparent by Photoshop or some other editing tool. (b) When the background image is drawn with the car image over it, the background can be seen through the windows.

Much generative art is dynamic, which is to say it involves motion and/or interaction, but many works are equivalent to paintings and drawings (*static*). *Glib* could be a tool for helping create these sorts of generative art works. Unlike other sorts of computer programs, those associated with art do not have a known predictable result that can be affirmed as correct or not. It is true that an artist begins with an idea of what their work should look like and what the message underlying it is, but paintings, sculptures, and generative works rarely finish the way they began.

So, either begin with an idea of what the image will look like or admit that the whole thing is an experiment and couch the idea in terms of a sentence or two. Here's one such sentence: "Imagine a collection of straight lines radiating from a set of randomly placed points within the drawing window, with each set of lines drawn in a saturated strong color."

Now an attempt would be made to create such an image using the functions that Glib offers. It is often the case that the first few tries are in error, but that one of them is interesting. An artist would pursue the interesting course instead of sticking to the original idea, of course. Here is an example: the code below was written with the idea that it would produce a collection of lines radiating from the point (400,600) from 0 degrees (horizontal right) to 180 degrees (horizontal left) with the color varying slightly:

```
r = 255
for i in range (1, 180, 2):
    stroke (r, 128, 128)
    line (400, 600, cos(i*conv)*500, sin(i*conv)*500)
    r = r - 0.5
```

The call to **line()** should have been:

```
line (400, 600, 400+cos(i*conv)*500, 600-sin(i*conv)*500)
```

so as to invert the Y coordinate. Instead, this created a much more interesting image. The code for one of the four loops in the final code is:

```
x = randrange(100, Width())
y = randrange (100, Height()-100)
r = 255
for i in range (1, 180, 2):
    stroke (128, r, 128)
    line (x, y, sin(i*conv)*500, cos(i*conv)*500)
    r = r - 0.5
```

Sometimes a small error can result in a more interesting result. This is rarely the case when writing scientific or commercial software.

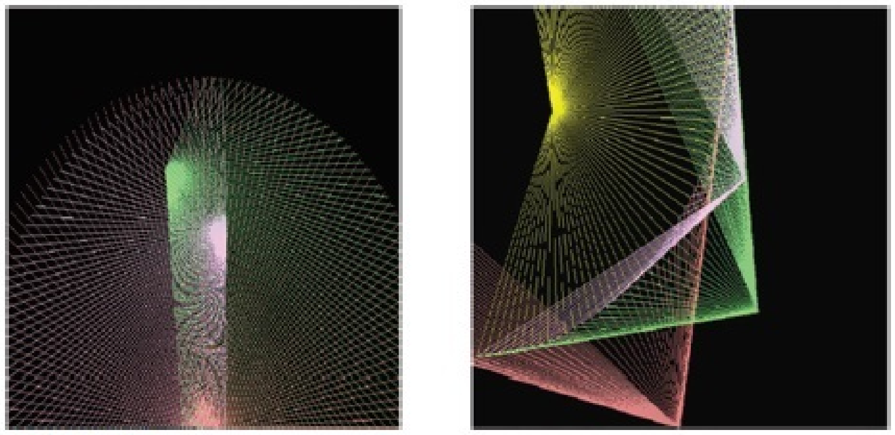


Figure 7.13
A generative artwork created partly by accident.

Generative art should be under the control of the artist, but does use random elements to add interest to the image. In the piece *Snow Boxes* by Noah Larsen a set of rectangles is drawn, but the specific size and location of these rectangles is random within constrained parameters. The overall color is also random within specified boundaries. Each rectangle is drawn as a collection of white pixels with a density that has been defined specifically for that rectangle so that the image consists of spatters of white pixels that can be identified as rectangular regions (Figure 7.14). Each time the program is executed a different image is created. The program for *Snow Boxes* was originally written in a language called *Processing*, but a Python version that uses *Glib* is:

```
# Snow boxes
# Original by Noah Larsen, @earlatron
```

```
from Glib import *
from random import *

startdraw (640, 480)
background(randrange(0,75), randrange(150,255),
randrange(0,75))

fill (255, 255, 255)
for i in range(0,10000):
    point (randrange(0,Width()), randrange(0,Height()))
for i in range(0,20):
    xs = randrange (0, Width())
    ys = randrange (0, Height())
    xe = randrange (xs, xs+randrange(30, 300))
    ye = randrange (ys, ys+randrange(30, 300))
    for j in range(0,10000):
        point (randrange(xs, xe+1),randrange(ys, ye+1))
enddraw()
```

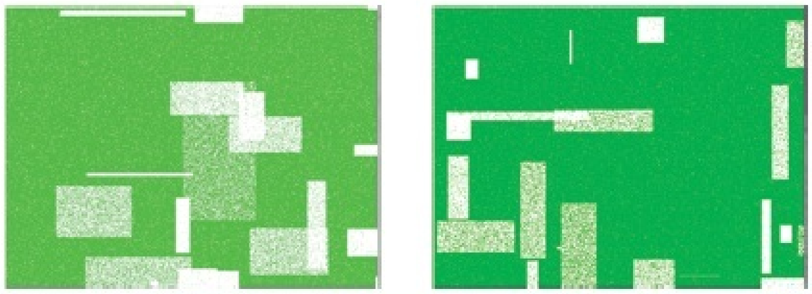


Figure 7.14
Output samples from the *Snow Boxes* program, examples of generative art.

7.2 SUMMARY

Since the advent of *Windows*, computer graphics has been assumed as a feature of a computer, but this has not always been true. Python does not have built-in features for doing graphics, but the standard user interface library *tkinter* does, and the library that accompanies this book, *Glib*, expands this and makes it more accessible. Drawing is accomplished by setting pixels within a drawing window to a desired color. Colors are specified by giving the amount of red, green, and blue that comprise the color.

Glib and most graphics libraries allow the user to draw lines, polygons, text, images, and to set pixels. These basic functions are combined by the programmer to create desired visualizations, such as histograms and pie charts.

Exercises

1. Write a Python program to create the image shown in Figure 7.15a. The image is grey, but the colors that are to be used to fill the circles are given as text. You need not include the text in your output.
2. Draw a set of 10 lines separated horizontally by 20 pixels, each parallel to the line specified by the end points (10, 20) and (200, 421). These lines may begin anywhere in the window.
3. Draw a pyramid using dark grey bricks (rectangles) as components. The base of the pyramid is to be 15 bricks horizontally, and each successive level is one brick smaller. (Figure 7.15b)
4. Draw a checkerboard. Each square should be 20x20 pixels, and the squares are red or yellow, alternating. A checkerboard is 8x8 squares.
5. Draw a triangle, a square, a regular pentagon, and a regular hexagon. Label these with their names as text above the shapes.
6. Write a program to draw a visual work in the visual style of Piet Mondrian's famous rectangular compositions, an example of which is shown in Figure 7.15d. Could triangular shapes be used instead of rectangles?
7. Modify the pie chart program so that the data is read from a file name "piein.dat."
8. Write a program that reads the file name of an image and displays the image in a window that is correctly sized.
9. The provided image named "digit.gif" contains some pixels that are pure red; that is, they have a pixel value of (255,0,0). Write a program that locates these pixels, draws a circle around them in a display of the image, and prints their x and y coordinates.
10. An *edge* in an image has the property that the pixel values on one side of the edge are significantly different (i.e., more than 40 levels) from those on the other side. Write a python program that reads an image and sets pixels at vertical edge locations to black and all other pixels to white; it then displays the result in a window. *Hints*: convert the image to grey or select one color value for the edges; make a working copy of the image.

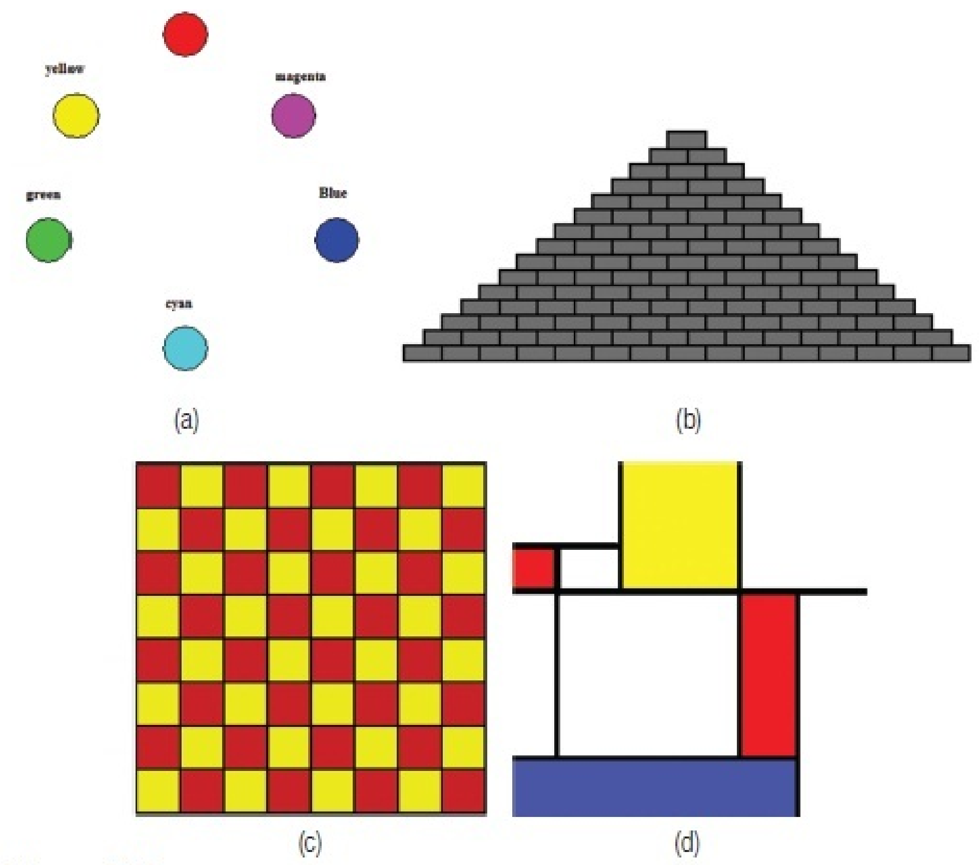


Figure 7.15
Figures to accompany the exercises.

Notes and Other Resources

Tkinter - Python interface to Tcl/Tk, *https://docs.python.org/3/library/tkinter.html*

Tkinter 8.5 reference. *http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html*

http://www.generativeart.com/

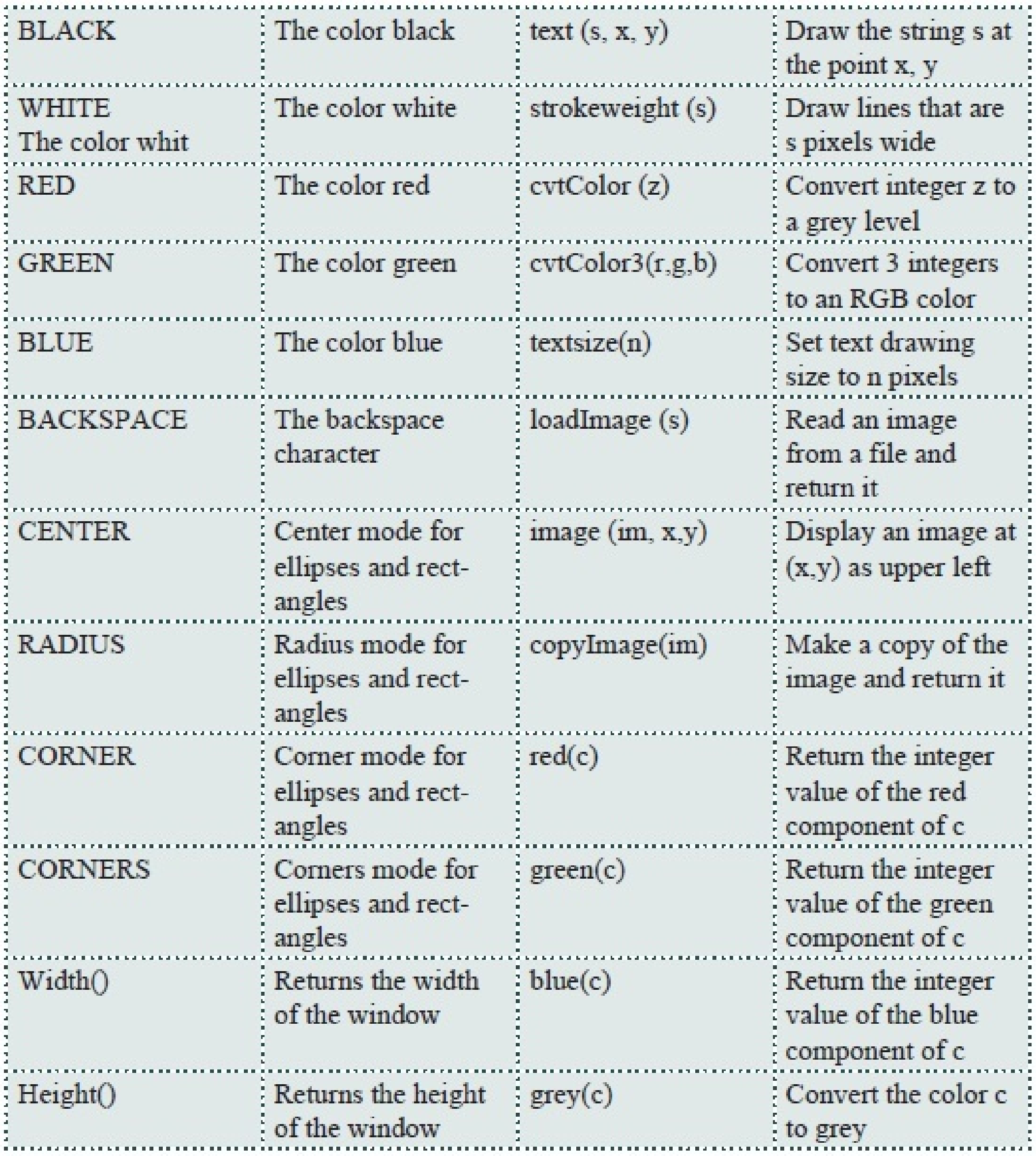
1. **John** F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. (2013). **Computer Graphics: Principles and Practice, 3rd Edition**, Addison-Wesley Professional.

2. Robin Landa, Rose Gonnella, and Steven Brower. (2006). **2D Visual Basics for Designers**, Delmar Cengage Learning.

3. Jeffrey McConnell. (2005). **Computer Graphics: Theory into Practice**, Jones & Bartlett Learning.

4. Matt Pearson. (2011). **Generative Art**, Manning Publications, ISBN-10: 1935182625.

Table 7.1
Glib functions and constants contained in the tkinter version.



BLACK	The color black	text (s, x, y)	Draw the string s at the point x, y
WHITE The color whit	The color white	strokeweight (s)	Draw lines that are s pixels wide
RED	The color red	cvtColor (z)	Convert integer z to a grey level
GREEN	The color green	cvtColor3(r,g,b)	Convert 3 integers to an RGB color
BLUE	The color blue	textsize(n)	Set text drawing size to n pixels
BACKSPACE	The backspace character	loadImage (s)	Read an image from a file and return it
CENTER	Center mode for ellipses and rect-angles	image (im, x,y)	Display an image at (x,y) as upper left
RADIUS	Radius mode for ellipses and rect-angles	copyImage(im)	Make a copy of the image and return it
CORNER	Corner mode for ellipses and rect-angles	red(c)	Return the integer value of the red component of c
CORNERS	Corners mode for ellipses and rect-angles	green(c)	Return the integer value of the green component of c
Width()	Returns the width of the window	blue(c)	Return the integer value of the blue component of c
Height()	Returns the height of the window	grey(c)	Convert the color c to grey

fill(r,g,b)	Set fill color	getpixel(im,i,j)	Get the pixel value (a color) at image pixel (i,j)
fill(g)	Set fill grey value	setpixel(im,i,j,c)	Set the pixel at image location (i,j) to color c
stroke (r, g, b)	Set line and outline color	save (im, s)	Save the image to a file named by the string s
stroke (g)	Set grey level for lines and outlines	background2(g)	Set the background color to grey value g
nostroke()	Turn off outline drawing	background(r,g,b)	Set the background color to (r,g,b)
ellipsemode(z)	Set ellipse drawing mode to where z is one of the following: CENTER, RADIUS, CORNER, or CORNERS	rectmode(m)	Set rectangle drawing mode to one of the following: CENTER, RADIUS, CORNER, or CORNERS
ellipse(x, y, w, h)	Draw an ellipse at x,y with specified width and height	rect (x1,y1, x2,x2)	Draw a rectangle at the given coordinates using the current mode
line(x0,y0,x1,y1)	Draw a line between the two coordinates	imageSize(s)	Examine the image file named by s and determine its size in pixels. Return a tuple (width, height)
point(x, y)	Draw a point (pixel) at x,y	arc (x0,y0,x1,y1,start, angle, s)	Draw an arc. See the description in this chapter.
triangle (x0,y0, x1,y1, x2,y2)	Draw a tringle between the three points	startdraw (width, height)	Begin drawing in an area having the specified size
		enddraw()	Stop calling Glib functions and display what was drawn